CST Part IB Supervisions

Example Sheet 2

Petar Veličković

Lent Term 2016

Compiler Construction

The course kicks off with a revision of automata theory and its application to lexing and parsing. One significant difference from previous years is that nontrivial parsing methods are done immediately at the start rather than the end of the course. For now, I will focus on lexing and basic CFG theory, leaving parsing for next week.

1. Why is it appropriate to define lexical tokens with regular expressions?

2. Provide a deterministic finite automaton that recognises these lexical tokens, given in priority order:

   `[:=, :, <numb>, <id>]`

   where `<numb>` is an integer, and `<id>` is a string.

3. Present a formal definition of a *grammar*. Define what it means for a grammar to be *context-free*. Prove that there exist context-free grammars which cannot be recognised by regular expressions[*].

4. What does it mean for a grammar to be *ambiguous*? Demonstrate ambiguity in the following grammars:

   a) $E \longrightarrow 1 \mid E - E$

   b)
   $$\begin{aligned} S &\longrightarrow A\ B \\ A &\longrightarrow a \mid a\ c \\ B &\longrightarrow b \mid c\ b \end{aligned}$$

   c)
   $$\begin{aligned} S &\longrightarrow a\ T\ b \mid T\ T \\ T &\longrightarrow a\ b \mid b\ a \end{aligned}$$

   d) $C \longrightarrow if\ E\ then\ C\ else\ C \mid if\ E\ then\ C$

[*]Interesting read after solving this exercise: https://pbs.twimg.com/media/CAja2roUYAAsBXP.png

5. Present a non-ambiguous context-free grammar for arithmetic expressions involving integers, addition (+) and multiplication ($*$), taking care of correctly encoding operator precedence.

6. Is HTML a context-free language?

Practical work

We will now delve into constructing a practical compiler for a toy language, one step at a time.

Our language is a simple imperative language with the following syntax specification:

- $x, y \in Var$ (variable identifiers; alphanumerical strings starting with a letter)
- $n \in Num$ (integer constants)
- $OPa \longrightarrow + \mid *$ (arithmetic operators)
- $OPb \longrightarrow and \mid or$ (boolean operators)
- $OPr \longrightarrow > \mid < \mid =$ (relations)
- $Aexp \longrightarrow n \mid x \mid Aexp\ OPa\ Aexp \mid (Aexp)$ (arithmetic expressions)
- $Bexp \longrightarrow true \mid false \mid !Bexp \mid Aexp\ OPr\ Aexp \mid Bexp\ OPb\ Bexp \mid (Bexp)$ (boolean expressions)
- $S \longrightarrow skip \mid x := Aexp \mid S; S \mid if\ Bexp\ then\ S\ else\ S \mid while\ Bexp\ do\ S \mid \{S\}$ (statements)

The natural first step is writing a lexer.

1. Implement a suitable data type/structure for encoding all of the possible tokens of the language.

2. Implement a function `lex` which takes in a path to a file and returns a list of tokens within the file.

3. Demonstrate that your function works as expected by presenting the result of lexing the following program:

```
b := 1;
if (b > 3)
then { a := 0 }
else { a := b + 1 }
```